Call-by-name, call-by-value, call-by-need and the linear lambda calculus

ELSEVIER

Theoretical Computer Science 228 (1999) 175–210

Theoretical Computer Science

www.elsevier.com/locate/tcs

Call-by-name, call-by-value, call-by-need and the linear lambda calculus

J. Maraist [a,*], M. Odersky [a], D.N. Turner [b], P. Wadler [c]

[a] *School of Computer and Information Science, University of South Australia, Warrendi Road, The Levels, South Australia 5095, Australia*

[b] *An Teallach Limited, Technology Transfer Centre, Kings Buildings, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK*

[c] *Bell Laboratories, Lucent Technologies, 700 Mountain Ave., Room 2T-304, Murray Hill, NJ 07974-0636, USA*

Abstract

Girard described two translations of intuitionistic logic into linear logic, one where $A \to B$ maps to $(!A) \multimap B$ and another where it maps to $!(A \multimap B)$. We detail the action of these translations on terms and show that the first corresponds to a call-by-name calculus, while the second corresponds to call-by-value. We further show that if the target of the translation is taken to be an affine calculus, where! controls contraction but weakening is allowed everywhere, then the second translation corresponds to a call-by-need calculus, as recently defined by Ariola, Felleisen, Maraist, Odersky and Wadler. Thus the different calling mechanisms can be explained in terms of logical translations, bringing them into the scope of the Curry–Howard isomorphism. Our results extend neatly to translations of extensions for recursion in the call-by-name and call-by-value calculi, and in general to extensions for products and for the corresponding untyped systems.

1. Introduction

Gordon Plotkin, in "Call-by-name, call-by-value and the λ-calculus" [30], demonstrated how two different calling mechanisms could be explained by two different translations into the continuation-passing style. At the time Plotkin wrote, the call-by-value translation was widely appreciated, but the call-by-name translation was less well known. In particular, the call-by-value translation was rediscovered several times (as related by Reynolds [31]), while the call-by-name translation appears to have been known only to Plotkin and Reynolds (the former credits it to the latter).

* Corresponding author. E-mail: maraist@cs.unisa.edu.au.

0304-3975/99/$ - see front matter
PII: S0304-3975(98)00358-2

While we hesitate to compare our work to Plotkin's, our goal here is somewhat similar. We demonstrate how the two different calling mechanisms can be explained by two different translations into linear logic. At the time we are writing, the call-by-name translation is widely appreciated, but the call-by-value translation is less well known. Both translations can be found in the original paper of Girard [14], the first based on mapping $A \to B$ into $(!A) \multimap B$ and the second based on mapping it into $!(A \multimap B)$. Girard devotes several pages to the first translation and less than a paragraph to the second, stating that "its interest is limited". That the first translation corresponds to call-by-name appears to be widely known, while the knowledge that the second translation corresponds to call-by-value appears to be restricted to a narrower circle.

A number of different lambda calculi based on linear logic have been proposed, including work by Lafont [19], Holmström [17], Wadler [39–42], Abramsky [1], Mackie [22], Lincoln and Mitchell [21], Troelstra [37], Benton, Bierman, de Paiva and Hyland [9–12] and della Rocca and Roversi [32]. Various embeddings of intuitionistic logic into linear logic have also been studied, including work by Girard [14], Troelstra [37] and Schellinx [35]. The linear lambda calculus used in this paper is a minor refinement of one previously presented by Wadler [41, 42], which is based on Girard's successor to linear logic, the Logic of Unity [15]. A similar calculus has been devised by Plotkin and Barber [6].

In many presentations of logic a key role is played by the structural rules: contraction provides the only way to duplicate an assumption, while weakening provides the only way to discard one. In linear logic [14], the presence of contraction or weakening is revealed in a formula by the presence of the 'of course' connective, written '!'. The Logic of Unity [15] takes this separation one step further by distinguishing linear assumptions, which one cannot contract or weaken, from nonlinear or intuitionistic assumptions, which one can.

Corresponding to Girard's first translation we define a mapping $\circ$ from the call-by-name to the linear calculus and show that this mapping is sound, in that $M \underset{\mathsf{Name}}{\twoheadrightarrow} N$ implies $M^\circ \underset{\mathsf{Lin}}{\twoheadrightarrow} N^\circ$, and complete, in that the converse also holds. Corresponding to Girard's second translation we define a second mapping $*$ from the call-by-value calculus to the linear calculus and show that this mapping is also sound, in that if $M \underset{\mathsf{Val}}{\twoheadrightarrow} N$ then $M^* \underset{\mathsf{Lin}}{\twoheadrightarrow} N^*$, but not complete. To recapture completeness of the second translation, we consider translations of standard reduction sequences, which are essentially the same as the evaluation machines of Plotkin's original presentation.

Mackie [23] has also observed that the first translation corresponds to call-by-name and the second to call-by-value. (He also states that these observations are common in the literature, but he gives no references and we have been unable to locate any.) Mackie's work is complementary to our own. Our translations are into a linear lambda calculus, corresponding to intuitionistic linear logic, while Mackie's translation is into proof nets, corresponding to classical linear logic. We prove soundness and completeness for beta (but not eta); while Mackie proves soundness (but not completeness) for beta and eta. Mackie also says nothing about call-by-need, which we do discuss.

Gonthier et al. [16] also have a translation based on taking $A \to B$ into $!(A \multimap B)$, but it is rather more complex than our translation at the term level, because it deals with the rather more complex notion of optimal reduction.

Our original motivation for studying these questions came from an interest not in call-by-name or in call-by-value, but instead in a call-by-need calculus, which was recently proposed by Ariola et al. [3, 4, 27]. The call-by-name calculus is not entirely suitable for reasoning about functional programs in lazy languages, because the beta rule may copy the argument of a function any number of times. The call-by-need calculus uses a different notion of reduction, observationally equivalent to the call-by-name calculus. But call-by-need, like call-by-value, guarantees that the argument to a function is not copied before it is reduced to a value.

The emphasis on avoiding copying suggests that the 'resource conscious' approach of linear logic may be relevant. In the linear lambda calculus (as in linear logic), the '!' connective is used to control duplication (contraction) and discarding (weakening) of lambda terms (proofs). For call-by-need we wish to avoid duplication but not discarding, so an appropriate target for our translation is an affine calculus, in which contraction is controlled by the ! connective but weakening is allowed everywhere. The use of distinct !-prefixes to control contraction and weakening separately has been studied first by Jacobs [18] for the full logic, and later by Maraist [24, 25] for a lambda calculus.

We derive the call-by-need calculus from the call-by-value calculus in two steps. The first step adds 'let' terms, which enforce sharing, to the call-by-value calculus. The resulting call-by-let calculus is observationally equivalent to call-by-value; the $*$ translation, easily extended, is still sound and complete for standard reduction. We then add one further law, which allows a value bound by a 'let' to be discarded without first being computed if the value is not needed for the result. The resulting call-by-need calculus is observationally equivalent to call-by-name as opposed to call-by-value; the $*$ translation remains sound and complete for standard reduction if its target is taken to be an affine calculus as opposed to the linear calculus.

As a result, the call-by-value and call-by-need calculi are brought into the scope of the Curry-Howard isomorphism, as the $*$ translation relates these to reductions of the linear calculus that have a clear logical explanation. An additional contribution of this work is that we confirm that our linear and affine calculi are confluent, and that they have notions of standard reduction. Although many linear calculi have been described, relatively few possess claims of confluence, notable exceptions being the work of Benton [9], Bierman [11] and della Rocca and Roversi [32]. It is also relatively uncommon to find notions of standard reduction in linear calculi as we present here.

As an application of these results, we have devised a type system that can infer information about which variables are used linearly in a call-by-need or call-by-value lambda calculus. Such types are useful for program transformation: the reduction $((\lambda x.M)N) \to M[x := N]$ does not in general hold for a call-by-value or call-by-need calculus, but it does hold if x is used linearly. It may also be useful for

implementing call-by-need: normally a closure needs to be overwritten on evaluation, but this step may be saved if the closure is bound to a variable that is used linearly. These applications are developed in a companion paper by Turner et al. [38].

We first presented these soundness and completeness results for general, but not standard, reduction sequences, in an earlier conference presentation [26] under a slightly different syntax. Our first presentation reported incorrectly that the $*$ translation is complete for general reduction sequences, which we have corrected here. The standard reduction results for Lin and Aff, as well as the present proof of conservative extension of Val to Let, are from the first author's thesis [24]. Our results now also apply to translations of the equality and observational equivalence relations of the various calculi, and to extended translations from the usual recursive extensions of call-by-name and call-by-value into a similarly extended recursive linear calculus. We have expanded the justification of some results, in particular spelling out some more details of the untyped systems.

The remainder of this paper is organised as follows. Section 2 introduces the linear lambda calculus. Sections 3–6 describe the call-by-name, call-by-value, call-by-let, affine and call-by-need calculi and their translations. Section 7 extends the results from the reduction theories to the equality and observational equivalence theories. Section 8 sketches how this work may be extended by adding products, by adding constants and primitive operations, or by removing types; and it remarks that adding sums or recursion is more problematic. Finally, Section 9 concludes with a discussion of some open questions.

2. The linear lambda calculus

Fig. 1 presents the details of the linear lambda calculus Lin. A type, corresponding to a formula of the logic, is either the base type, an 'of course' type or a linear function; we take Z to range over base types and take A, B, C to range over all types. A term, corresponding to a proof in the logic, is either a variable, an 'of course' introducer, an 'of course' eliminator, a function abstraction or a function application; we let x, y, z range over variables and L, M, N range over terms. We write $M[x := N]$ for the result of substituting term N for every free occurrence of the variable x in term M. A context is a term with a single hole [ ] which may be filled by another term. Note that substitution may not bind free variables, but hole-filling may in fact do so.

Typing environments are sets containing assumptions of the form $x : A$. We let Φ and Ψ range over intuitionistic assumptions, and Γ and Δ range over linear assumptions. In a judgement

$$\Phi;\ \Gamma \vdash M : A$$

the environments Φ and Γ should bind disjoint sets of identifiers. If Γ and Δ are environments with distinct variables, then Γ, Δ denotes their union; similarly for Φ and

Syntactic domains

Types	A,B,C	$::= Z \mid !A \mid A \multimap B$
Terms	L,M,N	$::= x \mid !M \mid$ let $!x = M$ in $N \mid \lambda x.M \mid M\,N$
Evaluation contexts	E	$::= [\,] \mid E\,M \mid$ let $!x = E$ in M
Answers	R	$::= \lambda x.M \mid !M$

Typing judgements

$$\frac{}{-;\ x:A \vdash x:A}\ \mathsf{Id} \qquad \frac{\Phi, y:A, z:A;\ \Gamma \vdash M:B}{\Phi, x:A;\ \Gamma \vdash M[y:=x, z:=x]:B}\ \mathsf{Contraction}$$

$$\frac{\Phi;\ \Gamma \vdash M:B}{\Phi, x:A;\ \Gamma \vdash M:B}\ \mathsf{Weakening} \qquad \frac{\Phi;\ x:A, \Gamma \vdash M:B}{\Phi, x:A;\ \Gamma \vdash M:B}\ \mathsf{Dereliction}$$

$$\frac{\Phi;\ - \vdash M:A}{\Phi;\ - \vdash !M:!A}\ !\text{-I} \qquad \frac{\Phi;\ \Gamma \vdash M:!A \quad \Psi, x:A;\ \Delta \vdash N:B}{\Phi, \Psi;\ \Gamma, \Delta \vdash \text{ let } !x = M \text{ in } N:B}\ !\text{-E}$$

$$\frac{\Phi;\ \Gamma, x:A \vdash M:B}{\Phi;\ \Gamma \vdash \lambda x.M:A \multimap B}\ \multimap\text{-I} \qquad \frac{\Phi;\ \Gamma \vdash M:A \multimap B \quad \Psi;\ \Delta \vdash N:A}{\Phi, \Psi;\ \Gamma, \Delta \vdash M\,N:B}\ \multimap\text{-E}$$

Reduction relation

$(\beta\multimap)$	$(\lambda x.M)N$	$\to M[x:=N]$
$(\beta!)$	let $!x = !N$ in M	$\to M[x:=N]$
$(!\multimap)$	(let $!x = L$ in M)N	$\to$ let $!x = L$ in (MN)
$(!!)$	let $!y =$ (let $!x = L$ in M) in N	$\to$ let $!x = L$ in (let $!y = M$ in N)

Standard reduction relation

$(\beta\multimap)$	$(\lambda x.M)N$	$\mapsto M[x:=N]$
$(\beta!)$	let $!x = !N$ in M	$\mapsto M[x:=N]$

Fig. 1. The linear lambda calculus Lin.

Ψ. We write '–' for an empty environment. We take environments to be sets rather than lists, so the order of bindings in an environment is irrelevant, and we may safely omit Exchange rules.

A typing judgement $\Phi;\ \Gamma \vdash M:A$ indicates that in the intuitionistic typing environment Φ plus linear environment Γ, term M has type A. If the judgement

$$x_1:A_1, \ldots, x_m:A_m;\ y_1:B_1, \ldots, y_n:B_n \vdash M:C \tag{1}$$

holds then the free variables of M will be drawn from $x_1, \ldots, x_m$, each of which occurs any number of times, and $y_1, \ldots, y_n$, each of which occurs linearly.

Those readers familiar with linear logic or the Logic of Unity may observe that the following three statements are equivalent:

- There exist variables $x_1,\ldots,x_m, y_1,\ldots,y_n$ and term M such that (1) holds in Lin.
- The judgement $!A_1,\ldots,!A_m,\ B_1,\ldots,B_n \vdash C$ holds in linear logic.
- The judgement $B_1,\ldots,B_m;\ A_1,\ldots,A_n \vdash C$; ; holds for the Logic of Unity (without polarities). Note that we write intuitionistic assumption on the outside and linear assumptions on the inside, the opposite of Girard's convention.

There are five rules concerned with the ! connective: Dereliction, Contraction and Weakening are structural rules, while !-I introduces and !-E eliminates the ! connective. An intuitionistic assumption can only be introduced by the !-E rule. The only thing that we may do with such an assumption is to duplicate it via Contraction, discard it via Weakening or convert it to a linear assumption via Dereliction. Furthermore, it is only intuitionistic assumptions that can appear in the !-I rule, as the empty linear typing environment indicates. Analogous to the conclusion $-;\ x:A \vdash x:A$ of rule Id, we can derive the conclusion $x:A;\ - \vdash !x:!A$ by combining Id, Dereliction and !-I.

The reduction relation is specified by two beta rules, $(\beta\multimap)$ and $(\beta!)$, and two commuting rules, $(!\multimap)$ and $(!!)$. We take the reduction relation to be the compatible closure of the given rules, as we will do for all reduction systems presented in the remainder of this paper. Also, in order to avoid name capture, we assume free and bound variables of a term to be distinct; for instance, in rules $(!\multimap)$ and $(!!)$, variable x cannot appear free in term N. We take the reduction relation to be *compatibly closed* under all contexts, that is, whenever $M \to N$, then $C[M] \to C[N]$ as well.

Some notation: we write $\twoheadrightarrow$ for the reflexive and transitive closure of $\to$, we write $=$ for the reflexive, symmetric and transitive closure of $\to$, and we write $\equiv$ for syntactic identity. When necessary, we may write the name of a calculus below a symbol to disambiguate, as in $\vdash_{\mathsf{Lin}}$ or $\underset{\mathsf{Lin}}{\twoheadrightarrow}$. We may also write the name of a rule below an arrow to indicate which rule is applied, as in $\underset{(\beta\multimap)}{\longrightarrow}$.

Each of the reduction rules has a logical basis.

- Rule $(\beta\multimap)$ arises when a $\multimap$-I rule meets a $\multimap$-E rule and the two rules annihilate.
- Rule $(\beta!)$ arises when a !-I rule meets a !-E rule and the two rules annihilate.
- Rule $(!\multimap)$ arises when a !-E rule meets a $\multimap$-E rule, commuting one through the other.
- Rule $(!!)$ arises when two !-E rules meet, commuting one through the other.

It is important to verify that the substitutions respect the restrictions on variables. In rule $(\beta\multimap)$ the variable x appears linearly in M, so any free variable that appears linearly in N will still appear linearly in $M[x:=N]$. Hence the substitution is well formed. In rule $(\beta!)$ the variable x may appear any number of times in N, so a free variable of M may be copied arbitrarily many times in $N[x:=M]$. It is here that distinguishing the two sorts of assumptions is helpful: the constraint on the !-I rule guarantees that the term $!M$ may only contain free variables that can appear any number of times. Hence this substitution is also well formed. The following lemma makes these ideas concrete. Since we express well-formedness via the type system, the

lemma says that the term which results from substituting a well-typed term for a free variable in some well-typed term is again well-typed.

Lemma 1 (Well-typed substitutions). (i) *Let* $\Phi;\Gamma,x:A\vdash M:B$ *and* $\Psi;\Delta\vdash N:A$. *Then* $\Phi,\Psi;\Gamma,\Delta\vdash M[x:=N]:B$.

(ii) *Let* $\Phi,x:A;\ \Gamma\vdash M:B$ *and* $\Psi;\ -\vdash N:A$. *Then* $\Phi,\Psi;\ \Gamma\vdash M[x:=N]:B$.

The proof of this lemma is trivial.

Some terminology: a calculus satisfies the *subject reduction* property if whenever $\Gamma\vdash M:A$ and $M\to N$ then $\Gamma\vdash N:A$. A calculus is *confluent* if whenever $L\twoheadrightarrow M$ and $L\twoheadrightarrow M'$, there exists a term N such that $M\mapsto N$ and $M'\twoheadrightarrow N$. A reduction relation $\mapsto\subseteq\to$ is a *standard strategy* for $\to$-reduction to a notion of answers R if (1) for all M there is at most one N such that $M\mapsto N$, (2) for all answers R we have $R\not\mapsto$, and (3) if $M\twoheadrightarrow R$ then there exists some R_0 such that $M\mapsto^{*}R_0$. All of the systems we study will possess both the subject reduction and confluence properties, and we will identify a notion of standard reduction for each.

These reduction rules are compatible with an operational interpretation where one evaluates (let $!x=M$ in N) by first evaluating M to the form $!M'$ and then evaluating $N[x:=M']$. Thus, we may view the term associated with !-E as forcing evaluation and the term associated with !-I as suspending evaluation. The standard reduction relation $\mapsto$ reflects this intuition. Standard reduction uses only the two beta rules, and is compatibly closed only under evaluation contexts rather than all contexts,

$$\frac{M\mapsto N}{E[M]\mapsto E[N]}.$$

We adapt the same notation for $\mapsto^{*}$, the transitive, reflexive closure of $\mapsto$, and for standard reduction by particular rules.

Lemma 2. *If* $M\underset{\mathsf{Lin}}{\longrightarrow}N$, *then*

(a) $L[x:=M]\underset{\mathsf{Lin}}{\twoheadrightarrow}L[x:=N]$, *and*

(b) $M[x:=L]\underset{\mathsf{Lin}}{\longrightarrow}N[x:=L]$.

Proposition 3. Lin *satisfies subject reduction.*

Proposition 4. Lin *is confluent.*

Proposition 5. *Reduction by* $\underset{\mathsf{Lin}}{\longmapsto}$ *is a standard strategy for* $\underset{\mathsf{Lin}}{\longrightarrow}$*-reduction to answers.*

Proof. Lemma 2 holds by structural inductions over contexts. The proof of Proposition 3 is straightforward, its essence being the logical content of the reduction rules listed above. We must make some technical accounting for moving typing

rules not reflected in terms' structure to a position "above" other rules which are so reflected: briefly, all uses of Dereliction and Weakening may be moved toward the leaves of the type tree, uses of Contraction on pairs of variables from different subterms may be kept toward the root, and uses of Contraction of pairs of variables from the same subterm may be moved toward the leaves. These commutings are easily shown by simple structural inductions.

The result for each of the four rules is similar. For ($\beta\multimap$) applied to a well-typed term, we have $\Phi;\ \Gamma \vdash (\lambda x.M)\ N : B$, and since we can clearly commute all but possibly some Contraction steps to above the ($\multimap$-E) step, we have an application of the ($\multimap$-E) rule leading to this first judgement by contractions, and similarly we can commute all other steps to above the ($\multimap$-I) step, and have a ($\multimap$-I) immediately above the ($\multimap$-E):

$$\dfrac{\dfrac{\dfrac{\vdots\ \mathscr{F}}{\Phi_0;\Gamma_0,x:A\vdash M_0:B}\ \multimap\text{-I}}{\Phi_0;\Gamma_0\vdash\lambda x.M_0:A\multimap B}\qquad\dfrac{\vdots\ \mathscr{G}}{\Psi_0;\Delta_0\vdash N_0:A}}{\Phi_0,\Psi_0;\Gamma_0,\Delta_0\vdash(\lambda x.M_0)N_0:B}\ \multimap\text{-E}.$$

$$\dfrac{\vdots}{\Phi;\Gamma\vdash(\lambda x.M)N:B}$$

Then with $\mathscr{F}$, $\mathscr{G}$ and Lemma 1(i) we have subject reduction. For rule (β!) we follow the same reasoning, building a type tree for a let-binding to a !-prefixed term rather than an application of an abstraction. For the (!$\multimap$, !!) rules, we need only observe that structural rules can be commuted above the relevant elimination steps, and do not need Lemma 1.

For both of Propositions 4 and 5 there is a variation on the usual notion of a marked redex which greatly facilitates proof: when marking commuting conversion rules we note not only the position of redexes, but also associate with the position a natural number counting the possible consecutive redexes. For example, in the term

$$\text{let } !x = (\text{let } !y = M_1 \text{ in let } !z = M_2 \text{ in } M_3) \text{ in } M_4,$$

we have only one present (!!) contraction, but the "let !x" prefix could take part in a second (!!) step immediately after the first:

$$\begin{aligned}&\text{let } !x = (\text{let } !y = M_1 \text{ in let } !z = M_2 \text{ in } M_3) \text{ in } M_4\\ &\quad\to \text{let } !y = M_1 \text{ in let } !x = (\text{let } !z = M_2 \text{ in } M_3) \text{ in } M_4\\ &\quad\to \text{let } !y = M_1 \text{ in let } !z = M_2 \text{ in let } !x = M_3 \text{ in } M_4\end{aligned}$$

By associating a number with these marks we can mark both redexes together. This ability may seem arcane, but it allows a useful notion of *developments*: a development is a reduction sequence which contracts only marked redexes. *Complete* developments end in unmarked terms, and the *residuals* of a set of marked redexes are those left over

by the contraction of other redexes. With only a simple marking scheme, without the associated numbers, marked reduction would not be confluent, and so different complete developments of the same term and marking could end in different unmarked terms. The extended marking scheme and its notion of developments are sufficient to prove the following technical result, which greatly simplifies the confluence and standardisation proofs:

Lemma 6. *All* Lin*-developments are finite, all can be extended to complete developments, and all complete* Lin*-developments of a given term and marking end in the same term. Moreover, given two different markings of the same underlying term, there exists a third marking such that the complete developments for each of the original two coincide with* (*partial*) *developments of the third.*

We have proven this result and analogous results for the other systems in this paper elsewhere [24]. The details of the marking scheme appear in a separate article [27], and we do not discuss this technical issue further.

Proposition 4 follows because any single reduction step can be seen as a complete development of a single redex; then the lemma allows an inductive argument for confluence. Proposition 5 follows from two observations: first, the marked-reduction theory allows reductions sequences to be reordered so that all standard steps may precede all non-standard steps, and secondly that if we have $M \to R$ by a non-standard step, then M is also an answer. □

We conclude this section with a few words about the relation of our linear lambda calculus to other formulations.

The formulation given here is based on the Logic of Unity [15], but omitting the extra complication of polarities. In Girard's presentation, the rule !-E does not appear, but it can be derived by combining his ! elimination rule (the fourth to last rule on the right in his Fig. 2) with one of his structural rules (the last rule on the right in his Fig. 1). We chose our formulation because it yields a simpler (β!) rule.

Our system follows the Logic of Unity, but differs from most other linear lambda calculi in that we use two forms of assumption, which enables the subject reduction property to be established in a simple way. The other systems listed in the introduction either lack this property altogether, or satisfy only a restricted version, or else possess full subject reduction but have a more complex syntax for ! introduction.

Also, most other systems treat Weakening and Contraction as logical rules with associated term forms. Our system treats Weakening and Contraction as structural rules, without the clutter of term forms. The result is more compact and arguably more suitable as the basis of a programming language.

Some elaboration of the above points can be found in our previous work [41, 42]. Every term of our language is also a term of the language in [41], from which it is easy to see how to give a semantics to this language in a categorical model in the style of Seely [36] as amended by Bierman [11, 12].

Syntactic domains

Types	A,B,C	$::= Z \mid A \to B$
Values	V	$::= x \mid \lambda x.M$
Terms	L,M,N	$::= V \mid MN$
Evaluation contexts	E	$::= [\,] \mid EM$
Answers	R	$::= \lambda x.M$

Typing judgements

$$\frac{}{x:A \vdash x:A}\ \text{Id} \qquad \frac{\Gamma \vdash M:B}{\Gamma, x:A \vdash M:B}\ \text{Weakening}$$

$$\frac{\Gamma, y:A, z:A \vdash M:B}{\Gamma, x:A \vdash M[y:=x, z:=x]:B}\ \text{Contraction}$$

$$\frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \to B}\to\text{-I} \qquad \frac{\Gamma \vdash M : A \to B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}\to\text{-E}$$

Reduction relation

$$(\beta\to) \quad (\lambda x.M)N \to M[x:=N]$$

Standard reduction relation

$$(\beta\to) \quad (\lambda x.M)N \mapsto M[x:=N]$$

Translation

Of types

$$Z^\circ \equiv Z$$
$$(A \to B)^\circ \equiv (!A^\circ) \multimap B^\circ$$

Of terms

$$x^\circ \equiv x$$
$$(\lambda x.M)^\circ \equiv \lambda y.\ \text{let}\ !x = y\ \text{in}\ M^\circ$$
$$(MN)^\circ \equiv M^\circ\ !N^\circ$$

Of typing environments

$$(x_1:A_1, \ldots, x_n:A_n)^\circ \equiv x_1:A_1^\circ, \ldots, x_n:A_n^\circ$$

Fig. 2. The call-by-name lambda calculus Name.

3. The call-by-name lambda calculus

Fig. 2 reviews the call-by-name lambda calculus Name and presents its translation into the linear lambda calculus. Types, terms and values are standard: a type is the base type or a function type, a term is a value or a function application, and a value is a variable or a function abstraction.

Environments are again sets of assumptions of the form $x:A$, where each variable x is distinct. A typing judgement $\Gamma \vdash M:A$ indicates that in environment Γ term M has type A. The type rules are standard. We chose a formulation with Weakening and Contraction to stress the connection with the linear type system, the key difference being that now the use of Contraction and Weakening is unconstrained.

There is a single reduction rule, $(\beta\rightarrow)$. As usual reduction $\rightarrow$ is closed under all contexts, and standard reduction $\mapsto$ is closed only under evaluation contexts. This calculus satisfies the usual subject reduction, confluence and standardisation results.

Proposition 7. Name *satisfies subject reduction.*

Proposition 8. Name *is confluent.*

Proposition 9. *Reduction by* $\underset{\text{Name}}{\longmapsto}$ *is a standard strategy for* $\underset{\text{Name}}{\longrightarrow}$*-reduction to answers.*

Translation $\circ$ takes types, terms and environments A, M and Γ of the call-by-name lambda calculus Name to types, terms and environments A°, M° and Γ° of the linear lambda calculus Lin. In the translation of abstractions, y is a fresh variable, not appearing in M.

The idea behind this translation is that all assumptions are taken as intuitionistic, and that ! is added to the left of $\multimap$, but not to the right. Every function argument is surrounded by !, which can be thought of as suspending evaluation, corresponding to the call-by-name discipline.

In particular, corresponding to $(\beta\rightarrow)$ we have

$$
\begin{aligned}
&((\lambda x.M)N)^\circ \\
&\quad \equiv \quad (\lambda y.\ \text{let } !x = y \text{ in } M^\circ)\ !N^\circ \\
&\quad \underset{(\beta\multimap)}{\longrightarrow} \text{let } !x = !N^\circ \text{ in } M^\circ \\
&\quad \underset{(\beta!)}{\longrightarrow} M^\circ[x := N^\circ] \\
&\quad \equiv \quad (M[x := N])^\circ
\end{aligned}
$$

which shows that the translation is sound.

Proposition 10 (Call-by-name translation). *The translation* $\circ$ *from* Name *to* Lin *preserves substitutions, typing judgements, reductions, standard reductions and equalities*:

(i) $(M[x := N])^\circ \equiv M^\circ[x := N^\circ]$.

(ii) $\Gamma \vdash_{\text{Name}} M : A$ *if and only if* $\Gamma^\circ \vdash_{\text{Lin}} M^\circ : A^\circ$.

(iii) $M \underset{\mathsf{Name}}{\twoheadrightarrow} N$ *if and only if* $M^\circ \underset{\mathsf{Lin}}{\twoheadrightarrow} N^\circ$.
(iv) $M \underset{\mathsf{Name}}{\mapsto\!\!\!\!\twoheadrightarrow} N$ *if and only if* $M^\circ \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\twoheadrightarrow} N^\circ$.

Proof. We prove (i) by an easy structural induction over terms of Name and prove (ii) by an easy structural induction over type derivations in Name. The proof of (iii) in the forward direction is given above, with compatible closure trivial as $[\,]^\circ \equiv [\,]$. For the backward direction, we consider the grammar

$$S, T ::= x \mid \lambda y.\ \text{let}\ !x = y\ \text{in}\ S \mid S!T \mid\ \text{let}\ !x = !S\ \text{in}\ T$$

restricted to terms well-typed according to the usual rules for Lin. This grammar defines the set of Lin terms reachable from translations of terms in Name: $M^\circ \underset{\mathsf{Lin}}{\twoheadrightarrow} S$. We then define an erasure $\dagger$ that takes this set back into Name:

$$c^\dagger \equiv c$$

$$x^\dagger \equiv x$$

$$(\lambda y.\ \text{let}\ !x = y\ \text{in}\ S)^\dagger \equiv \lambda x.S^\dagger$$

$$(S!T)^\dagger \equiv S^\dagger T^\dagger$$

$$(\text{let}\ !x = !S\ \text{in}\ T)^\dagger \equiv T^\dagger[x := S^\dagger].$$

Note in particular that the typing rules require that

$$(\forall S \equiv \lambda y.\ \text{let}\ !x = y\ \text{in}\ S_0)\ \ y \notin \mathrm{fv}(S_0),$$

insuring that $\dagger$ is meaningful for such S, and does not introduce a new free variable in $S^\dagger$ from an occurrence of y in S_0.

The following lemma is straightforward:

Lemma 11. *The inverse standard translation has the following properties*:
(a) *The mapping* $\dagger$ *is a right-inverse of* $\circ$: $M^{\circ\dagger} \equiv M$, *for all* M *in* Name.
(b) *The mapping* $\dagger$ *preserves substitutions*: $(S[x := T])^\dagger \equiv S^\dagger[x := T^\dagger]$.
(c) *The mapping* $\dagger$ *sends* Lin*-reduction sequences to* Name*-reduction sequences*: *if* $S \underset{\mathsf{Lin}}{\longrightarrow} T$ *then* $S^\dagger \underset{\mathsf{Name}}{\twoheadrightarrow} T^\dagger$.
(d) *The mapping* $\dagger$ *sends* Lin*-standard reduction sequences to* Name*-standard reduction sequences*: *if* $S \underset{\mathsf{Lin}}{\longmapsto} T$ *then* $S^\dagger \underset{\mathsf{Name}}{\mapsto\!\!\!\!\twoheadrightarrow} T^\dagger$.

Now, $M^\circ \underset{\mathsf{Lin}}{\twoheadrightarrow} N^\circ$ implies by (c) that $M^{\circ\dagger} \underset{\mathsf{Name}}{\twoheadrightarrow} N^{\circ\dagger}$, which implies by (a) that $M \underset{\mathsf{Name}}{\twoheadrightarrow} N$.

Clause (iv) follows from (d), and noticing that in the arguments for Clause (iii) we always map standard redexes to standard redexes and evaluation positions to evaluation positions. □

Syntactic domains As for Name, except

Evaluation contexts $E ::= [\,] \mid EM \mid (\lambda x.M)E$

Typing judgements As for Name.

Reduction relation

$(\beta\!\rightarrow_v) \quad (\lambda x.M)V \rightarrow M[x := V]$

Standard reduction relation

$(\beta\!\rightarrow_v) \quad (\lambda x.M)V \mapsto M[x := V]$

Translation

Of types

$$A^* \equiv !A^+$$
$$Z^+ \equiv Z$$
$$(A \rightarrow B)^+ \equiv (A^* \multimap B^*)$$

Of terms

$$V^* \equiv !V^+$$
$$(M\,N)^* \equiv (\text{let } !z = M^* \text{ in } z)N^*$$
$$x^+ \equiv x$$
$$(\lambda x.M)^+ \equiv \lambda y.\ \text{let } !x = y \text{ in } M^*$$

Of typing environments

$$(x_1 : A_1, \ldots, x_n : A_n)^* \equiv x_1 : A_1^+, \ldots, x_n : A_n^+$$

Fig. 3. The call-by-value lambda calculus Val.

4. The call-by-value lambda calculus

Fig. 3 reviews the call-by-value lambda calculus Val and presents its translation into the linear lambda calculus. Types, terms and values are as in the call-by-name calculus.

There is a single reduction rule, $(\beta\!\rightarrow_v)$, which is a restriction of $(\beta\!\rightarrow)$ to the case when the function argument is a value, which is reflected in an additional evaluation context form. Again, the usual subject reduction, confluence and standardisation results hold.

Proposition 12. Val *satisfies subject reduction.*

Proposition 13. Val *is confluent.*

Proposition 14. *Reduction by* $\underset{\mathsf{Val}}{\longmapsto}$ *is a standard strategy for* $\underset{\mathsf{Val}}{\longrightarrow}$*-reduction to answers.*

Translation $*$ takes types, terms and environments A, M and Γ of Val to types, terms and environments A^*, M^* and Γ^* of Lin. There is also an auxiliary mapping $+$ from types and values A and V of Val to types and terms A^+ and V^+ of Lin; this mapping omits the outermost '!'. As before, in the translation of applications and abstractions we impose the additional condition that y and z should be fresh variables not appearing in M.

The idea behind this translation is that ! is added on both the left and right of $\vdash$ and $\multimap$. Function arguments are no longer surrounded by !, so the argument will be reduced until a ! is encountered before evaluation of the function body proceeds. Under this translation, the only terms beginning with ! are those of the form $V^* \equiv !V^+$, so function arguments are reduced by linear reduction to a form corresponding to values in the original system.

In particular, corresponding to the call-by-name ($\beta\rightarrow$) rule we have

$$
\begin{aligned}
&((\lambda x.M)N)^* \\
&\quad \equiv \quad (\text{let } !z = !(\lambda y.\ \text{let } !x = y \text{ in } M^*) \text{ in } z)\ N^* \\
&\quad \underset{(\beta!)}{\longrightarrow} (\lambda y.\ \text{let } !x = y \text{ in } M^*)\ N^* \\
&\quad \underset{(\beta\multimap)}{\longrightarrow} \text{let } !x = N^* \text{ in } M^*,
\end{aligned}
$$

and the reduction cannot necessarily proceed further. But if we replace the argument term N by a value V, then corresponding to the call-by-value ($\beta\rightarrow_v$) rule we have

$$
\begin{aligned}
&((\lambda x.M)V)^* \\
&\quad \underset{\mathsf{Lin}}{\longrightarrow} \text{let } !x = !V^+ \text{ in } M^* \\
&\quad \underset{(\beta!)}{\longrightarrow} M^*[x := V^+] \\
&\quad \equiv \quad (M[x := V])^*
\end{aligned}
$$

which shows that the translation is sound.

Proposition 15 (Call-by-value translation). *The translation* $*$ *from* Val *to* Lin *preserves substitution of values, and preserves typing judgements, reductions, standard reductions and equalities.*

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \vdash_{\mathsf{Val}} M : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Lin}} M^* : A^*$, *and* $\Gamma \vdash_{\mathsf{Val}} V : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Lin}} V^+ : A^+$.

(iii) *If* $M \underset{\mathsf{Val}}{\twoheadrightarrow} N$ *then* $M^* \underset{\mathsf{Lin}}{\twoheadrightarrow} N^*$.
(iv) $M \underset{\mathsf{Val}}{\mapsto\!\!\!\!\twoheadrightarrow} N$ *if and only if* $M^* \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\twoheadrightarrow} N^*$.

Proof. The proof of (i) and (ii) are analogous to the corresponding proofs for Name. The proof of (iii) is given above. For soundness in (iv) it is clear that standard steps and evaluation contexts are preserved. For completeness we consider the standard reduction-closed images of Val terms S, T and right inverse $\ddagger$ of $*$, in the same manner as in the completeness result for Name:

Term image	$S, T ::= !U \mid P\ S \mid$ let $!x = S$ in T
Value image	$U ::= x \mid \lambda y.$ let $!x = y$ in S
Prefix	$P ::= U \mid$ let $!x = S$ in x

We consider only well-typed terms from these grammars. As right-inverse to $*$ we take

$$(!U)^\ddagger \equiv U^\ddagger$$
$$(P\ S)^\ddagger \equiv P^\ddagger S^\ddagger$$
$$(\text{let } !x = S \text{ in } T)^\ddagger \equiv (\lambda x.T^\ddagger)S^\ddagger$$
$$x^\ddagger \equiv x$$
$$(\lambda y.\text{ let } !x = y \text{ in } S)^\ddagger \equiv \lambda x.S^\ddagger$$
$$(\text{let } !x = S \text{ in } x)^\ddagger \equiv S^\ddagger$$

where for contexts $\ddagger$ simply preserves the hole.

Lemma 16. (a) *The mapping* $\ddagger$ *is a right-inverse of* $*$: $M^{\circ\ddagger} \equiv M$, *for all* $M \in \mathsf{Val}$.

(b) *The mapping* $\ddagger$ *preserves substitutions of value-images*: $(S[x := U])^\ddagger \equiv S^\ddagger[x := U^\ddagger]$.

(c) *The mapping* $\ddagger$ *sends* Lin *standard reduction sequences to* Name *standard reduction sequences*: *if* $S \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\twoheadrightarrow} T$ *then* $S^\ddagger \underset{\mathsf{val}}{\mapsto\!\!\!\!\twoheadrightarrow} T^\ddagger$.

The first two clauses of this lemma are straightforward. It is clear that $\ddagger$ preserves evaluation contexts formable from terms S, so for reduction, standard $\ddagger$-images are invariant under $(\beta\multimap)$ standard steps and top-level $(\beta!)$ contractions of prefixes P; $(\beta!)$ contraction of a term S translates to a $(\beta\to_v)$ step,

$$\begin{aligned}
&(\text{let } !x = !U \text{ in } S)^\ddagger \\
&\quad\equiv\quad (\lambda x.S^\ddagger)U^\ddagger \\
&\quad\underset{(\beta\to_v)}{\longrightarrow}\quad S^\ddagger[x := U^\ddagger] \\
&\quad\equiv\quad (S[x := U])^\ddagger. \qquad \square
\end{aligned}$$

However, the steadfast translation of Val-reduction is incomplete.

Example 17. Let I be the identity function $\lambda x.x$, P be any non-value, and M any term, all in Val. Then $I\,P\,M \underset{\mathsf{Val}}{\not\to} PM$ in general, but

$$
\begin{aligned}
&(I\,P\,M)^* \\
&\quad \equiv (\text{let } !z = ((\text{let } !w = !J \text{ in } w)P^*) \text{ in } z)\ M^* \\
&\quad \to (\text{let } !z = (JP^*) \text{ in } z)\ M^* \\
&\quad \to (\text{let } !z = (\text{let } !x = P^* = !x) \text{ in } z)\ M^* \\
&\quad \to (\text{let } !x = P^* \text{ in let } !z = !x \text{ in } z)\ M^* \\
&\quad \to (\text{let } !x = P^* \text{ in } x)\,M^* \\
&\quad \equiv (P\,M)^*
\end{aligned}
$$

5. The call-by-let lambda calculus

Fig. 4 defines the call-by-let lambda calculus Let and presents its translation into the linear lambda calculus. The terms are the same as previously, plus a 'let' construct, and the values remain unchanged. The types and type rules are also the same, with the addition of the obvious rule for 'let'.

Reduction for Let is defined by the four rules (I, V, C, A), which abbreviate *introduce*, *value*, *commute* and *associate*. The evaluation contexts require evaluation of let-bound terms rather than arguments, although given the (I) rule this difference is trivial. As before, subject reduction, confluence and standardisation both hold.

Proposition 18. Let *satisfies subject reduction.*

Proposition 19. Let *is confluent.*

Proposition 20. *Reduction by* $\underset{\mathsf{Let}}{\longmapsto}$ *is a standard strategy for* $\underset{\mathsf{Let}}{\longrightarrow}$ *reduction to values.*

Proof. Subject reduction can be verified by structural induction on the contractum as for the preceding calculi. Confluence and standardisation are straightforward by the same techniques described above for Lin [24, 27]. □

The translation $*$ is extended by adding a clause for let-bindings.

Proposition 21 (Call-by-let translation). *The translation* $*$ *from* Let *to* Lin *preserves substitutions of values and preserves typing judgements, reductions and standard reductions.*

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \vdash_{\mathsf{Let}} M : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Lin}} M^* : A^*$, *and* $\Gamma \vdash_{\mathsf{Let}} V : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Lin}} V^+ : A^+$.

Syntactic domains As for Val, plus the following:

Terms	L, M, N	$::= \cdots \mid \text{let } x = M \text{ in } N$
Evaluation contexts	E	$::= \cdots \mid \text{let } x = E \text{ in } M$

Typing judgements As for Val, plus the following:

$$\frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : B}{\Gamma, \Delta \vdash \text{let } x = M \text{ in } N : B}\,\mathsf{Let}$$

Reduction relation

(I)	$(\lambda x.M)N$	$\to \text{let } x = N \text{ in } M$
(V)	$\text{let } x = V \text{ in } M$	$\to M[x := V]$
(C)	$(\text{let } x = L \text{ in } M)N$	$\to \text{let } x = L \text{ in } (MN)$
(A)	$\text{let } y = (\text{let } x = L \text{ in } M) \text{ in } N$	$\to \text{let } x = L \text{ in } (\text{let } y = M \text{ in } N)$

Standard reduction relation

(I)	$(\lambda x.M)N$	$\mapsto \text{let } x = N \text{ in } M$
(V)	$\text{let } x = V \text{ in } M$	$\mapsto M[x := V]$

Translation As for Val, plus the following:

$$(\text{let } x = M \text{ in } N)^* \equiv \text{let } !x = M^* \text{ in } N^*$$

Fig. 4. The call-by-let calculus Let.

(iii) *If* $M \underset{\mathsf{Let}}{\twoheadrightarrow} N$ *then* $M^* \underset{\mathsf{Lin}}{\twoheadrightarrow} N^*$.
(iv) $M \underset{\mathsf{Val}}{\mapsto\!\!\!\!\twoheadrightarrow} N$ *if and only if* $M^* \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\twoheadrightarrow} N^*$.

Proof. The proof of (i) and (ii) is similar to that for Val. To prove (iii) we consider each possible reduction in Let. Reduction by rule (I) translates to

$$\begin{aligned}
&(\text{let } z = !(\lambda y.\ \text{let } !x = y \text{ in } M^*) \text{ in } z)\ N^* \\
&\underset{(\beta!)}{\longrightarrow} (\lambda y.\ \text{let } !x = y \text{ in } M^*)\ N^* \\
&\underset{(\beta\multimap)}{\longrightarrow} \text{let } !x = N^* \text{ in } M^*.
\end{aligned}$$

Reduction by rule (V) translates to

$$\text{let } !x = !V^+ \text{ in } M^* \underset{(\beta!)}{\longrightarrow} M^*\{x := V^+\}.$$

Reduction by rule (C) translates to

$$\begin{aligned}&(\text{let } !z = (\text{let } !x = L^* \text{ in } M^*) \text{ in } z)\ N^* \\ &\quad\xrightarrow[(!!)]{} (\text{let } !x = L^* \text{ in } (\text{let } !z = M^* \text{ in } z))\ N^* \\ &\quad\xrightarrow[(!\multimap)]{} \text{let } !x = L^* \text{ in } ((\text{let } !z = M^* \text{ in } z)\ N^*).\end{aligned}$$

And reduction by rule (A) translates to $(!!)$ in Lin.

As usual soundness in Clause (iv) is the observation that the (I,V) cases above preserve standard steps and compatible closure under evaluation contexts. For completeness we can adapt the proof of this result for Val, taking S, T, U, P as there, and the same right-inverse $\ddagger$ except taking

$$(\text{let } !x = S \text{ in } T)^\ddagger \equiv \text{let } x = S^\ddagger \text{ in } T^\ddagger$$

This $\ddagger$ is a right inverse of $*$ on the extended language of Let-terms. Furthermore, an analysis of standard Lin-reductions shows that $\ddagger$ preserves both standard steps and compatible closure under evaluation contexts. □

The counterexample to completeness for the translation of Val-reduction in Example 17 applies to the translation of Let-reduction sequences as well.

What is the relationship between Val and Let? Clearly, every $\beta{\to_v}$-reduction in Val can be simulated in Let by a pair of (I) and (V) reductions. That is, Let-reduction extends Val-reduction, and in fact this extension is conservative.

Proposition 22. Let *conservatively extends* Val: *for all terms* M *and* N *in* Val, $M \underset{\text{Val}}{\twoheadrightarrow} N$ *if and only if* $M \underset{\text{Let}}{\twoheadrightarrow} N$.

Proof. Conservative extension requires some syntactic machinery which is complicated, though not especially deep. We summarise the result here; the full details appear in Maraist's thesis [24].

First, we can extend the marked reduction theory of Let terms to attach a simple mark to let-bindings, $^{[\prime]}\text{let } x = M \text{ in } N$, and consider alternate (C,A) rules (C_+,A_+) which add these markings to their results,

$$\begin{aligned}&(C_+)\quad (\text{let } x = L \text{ in } M)\ N \to {}^{[\prime]}\text{let } x = L \text{ in } MN \\ &(A_+)\quad \text{let } x = (\text{let } y = L \text{ in } M) \text{ in } N \to {}^{[\prime]}\text{let } y = L \text{ in } {}^{[\prime]}\text{let } x = M \text{ in } N.\end{aligned}$$

Then any set of (C,A) steps in a given term which do not overlap can be sensibly lifted to (C_+,A_+) steps. Since conservative extension in effect requires that we be able to do without (C,A) steps, we can use these marks to reorder reduction sequences to apply the (V) rule to these bindings before the (C,A) step, thus removing the need for the (C,A) step. A reduction step is *wicked* if it occurs in a context $C_0[{}^{[\prime]}\text{let } x = M \text{ in } C_1]$, and is otherwise *righteous*. The two main results on this marked system are the following.

Recall that one writes $(M,\mathscr{F}) \underset{\text{cpl}}{\longrightarrow} N$ to indicate that N is the result of the complete development of redexes $\mathscr{F}$ in M.

(i) If $\mathscr{F}$ is a set of (C_+,A_+) steps in a term M which do not overlap, the complete development of these redexes is N, and $\sigma : N \underset{(I,V)}{\twoheadrightarrow} L$ is a righteous (I,V) reduction sequence, then there exists some righteous (I,V)-reduction sequence σ from M such that $\sigma : M \underset{(I,V)}{\twoheadrightarrow} M_0$ and $(M_0,\mathscr{F}/\sigma) \underset{\text{cpl}}{\longrightarrow} L$.

(ii) Let $\sigma : M \underset{\mathsf{Let}}{\twoheadrightarrow} N$, and let N have no $[\prime]$-marked bindings. Then there exists some righteous τ such that $\tau : M \underset{\mathsf{Let}}{\twoheadrightarrow} N$.

The second main step for conservative extension is to notice that we can reorder the steps in any (I,V) sequence into two halves: first a sequence of $(I^?V)$ steps, which are (V) steps possibly preceded by the (I) step generating the let-binding contracted by the (V) step, and second a sequence of (I) steps. If a sequence ends in a Val-term, which has no let-bindings, then this (I) sequence is zero-length. Moreover, if we map Let terms to Val terms by taking let-bindings to applications,

$$\text{let } x = M \text{ in } N \rightsquigarrow (\lambda x.N)M,$$

then each $(I^?V)$ step maps to a $(\beta\rightarrow)$ step.

So for conservative extension we consider a reduction sequence $\sigma : M \underset{\mathsf{Let}}{\twoheadrightarrow} N$ where $M,N \in \mathsf{Val}$, and show by induction on the number of (C,A) steps in σ that $M \underset{(I,V)}{\twoheadrightarrow} N$. For the inductive step with a final (C,A) step from M_0 to M_1,

$$M \underset{\mathsf{Let}}{\twoheadrightarrow} M_0 \underset{(C,A)}{\longrightarrow} M_1 \underset{(I,V)}{\twoheadrightarrow} N.$$

We can lift the final (C,A) step to a (C_+,A_+) step $M_0 \underset{(C_+,A_+)}{\longrightarrow} M_1'$; then since $N \in \mathsf{Val}$ it has no bindings, and hence no $[\prime]$-marked bindings, so there is a righteous sequence from M_1' to N. Then by (i) above we have a sequence

$$M_0 \underset{(I,V)}{\longrightarrow} N_0 \twoheadrightarrow (C,A)N,$$

but since $N \in \mathsf{Val}$ has no bindings, we have $N_0 \equiv N$, and by the second main step $M_0 \underset{(I^?V)}{\twoheadrightarrow} N$ and $M_0 \underset{\mathsf{Val}}{\twoheadrightarrow} N$. The proposition follows by the induction hypothesis on $M \underset{\mathsf{Let}}{\twoheadrightarrow} M_0$. □

6. The call-by-need calculus

In the call-by-name translation, every function argument was surrounded by !, so each argument could be freely duplicated or discarded. In the call-by-value and call-by-let translations, only values are surrounded by !, so while non-values cannot be duplicated, they also cannot be discarded. The call-by-need calculus differs from these two in that any term may be discarded if it is not needed, but a term should not be

Syntactic domains As for Lin, except as follows:

Evaluation contexts	$E ::= [] \mid !E \mid EM \mid \text{let } !x = M \text{ in } E \mid \text{let } !x = E \text{ in } E_0[x]$
Answers	$R ::= \lambda x.M \mid !R \mid \text{let } !x = M \text{ in } R$

Typing judgements As for Lin, but changing Weakening as follows:

$$\frac{\Phi; \Gamma \vdash M : B}{\Phi; \Gamma, x : A \vdash M : B}\ \text{Weakening}$$

Reduction relation As for Lin, plus the following:

(!Weakening) $\text{let } !x = M \text{ in } N \to N, \quad \text{if } x \text{ is not free in } N$

Standard reduction relation

$(\beta\multimap_s)$	$(\lambda x.M)N$	$\mapsto M[x := N]$
$(\beta!_s)$	$\text{let } !x = !R \text{ in } E[x]$	$\mapsto (E[x])[x := R]$
$(!\multimap_s)$	$(\text{let } !x = M \text{ in } R)N$	$\mapsto \text{let } !x = M \text{ in } RN$
$(!!_s)$	$\text{let } !y = (\text{let } !x = M \text{ in } R) \text{ in } E[x]$	$\mapsto \text{let } !x = M \text{ in } (\text{let } !y = R \text{ in } E[x])$

Fig. 5. The affine lambda calculus Aff.

duplicated unless and until it has been reduced to a value. Thus, we wish to shift to a calculus where discarding (Weakening) is always allowed, but duplication (Contraction) remains under the strict control of the ! connective.

Fig. 5 presents the details of the resulting affine lambda calculus Aff. The types, terms and environments are the same as for the linear lambda calculus Lin. The type rules are also identical to those for Lin, with the exception that the rule Weakening of Lin, which allows weakening only on intuitionistic assumptions (to the left of the semicolon), is replaced by a rule which allows weakening on linear assumptions (to the right of the semicolon). This new rule is strictly stronger than the previous rule, which can be derived by combining the new Weakening rule with Dereliction.

Reduction is defined by the rules $(\beta\multimap, \beta!, !\multimap, !!)$ together with a new rule (!Weakening). Again, this rule has a logical basis.

– Rule (!Weakening) arises when a !-E rule meets a Weakening rule, commuting one through the other.

The (!Weakening) rule cannot be valid in Lin, as it does not satisfy subject reduction. For instance, $-;\ y : !A,\ z : B \vdash \text{let } !x = y \text{ in } z : B$, which is a valid judgement in both Lin and Aff, reduces to $-;\ y : !A,\ z : B \vdash z : B$, which is valid in Aff but not in Lin.

What is the operational impact of the switch to an affine calculus? Recall that in the linear calculus, one evaluates $(\text{let } !x = M \text{ in } N)$ by first evaluating M to the form

$!M'$ and then evaluating $N[x := M']$. This notion of evaluation is not suitable for the affine calculus, as it would violate the (!Weakening) rule. Instead, one must evaluate (let $!x = M$ in N) by binding M to a closure which is evaluated only if x is required during the evaluation of N. Moreover, rather than taking ! as suspending computation, we interpret it as an update indicator, reducing the term under it to an answer when we know that it will be needed. Thus, much of the call-by-need machinery is implicit in the (!Weakening) rule.

As before, the logical origin of the rules ensures the subject reduction property.

Furthermore, confluence and standardisation still hold.

Proposition 23. Aff *satisfies subject reduction.*

Proposition 24. Aff *is confluent.*

Proposition 25. *Reduction by* $\underset{\mathsf{Aff}}{\longmapsto}$ *is a standard strategy for* $\underset{\mathsf{Aff}}{\longrightarrow}$ *reduction to answers.*

Proof. By the same general techniques as for Lin. □

Fig. 6 defines the call-by-need calculus Need as a slight modification of the call-by-let calculus Let. The only change is the addition of a new reduction rule (G), which allows unneeded computations to be discarded. The name (G) abbreviates *garbage collection*. In contrast to Let-evaluation, in Need a let-bound term is evaluated only on demand for the variable to which it is bound.

Proposition 26. Need *satisfies subject reduction.*

Proposition 27. Need *is confluent.*

Proposition 28. *Reduction by* $\underset{\mathsf{Need}}{\longmapsto}$ *is a standard strategy for* $\underset{\mathsf{Need}}{\longrightarrow}$ *reduction to answers.*

Proof. Subject reduction is easy, requiring just one more case than Let. We have previously published proofs of confluence and standardisation for call-by-need [24, 27]. □

By design, if $*$ is now viewed as taking the call-by-need calculus Need into the affine calculus Aff, then the transformation result still holds.

Proposition 29 (Call-by-need translation). *The translation* $*$ *from* Need *to* Aff *preserves substitutions of values, and preserves typing judgements, reductions and standard reductions*:

(i) $(M[x := V])^* \equiv M^*[x := V^+]$.

(ii) $\Gamma \vdash_{\mathsf{Need}} M : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Aff}} M^* : A^*$, *and* $\Gamma \vdash_{\mathsf{Need}} V : A$ *if and only if* $\Gamma^* \vdash_{\mathsf{Aff}} V^+ : A^+$.

Syntactic domains As for Let, except as follows:

Evaluation contexts	E	$::= [] \mid E\,M \mid \text{let } x = M \text{ in } E \mid \text{let } x = E \text{ in } E[x]$
Answers	R	$::= \lambda x.M \mid \text{let } x = M \text{ in } R$

Typing judgements As for Let.
Reduction relation As for Let, plus the following:

$$(G) \quad \text{let } x = M \text{ in } N \to N, \qquad \text{if } x \text{ is not free in } N$$

Standard reduction relation

(I)	$(\lambda x.M)N$	$\mapsto$	$\text{let } x = N \text{ in } M$
(V)	$\text{let } x = V \text{ in } E[x]$	$\mapsto$	$(E[x])[x := V]$
(C)	$(\text{let } x = M \text{ in } R)N$	$\mapsto$	$\text{let } x = M \text{ in } (RN)$
(A)	$\text{let } y = (\text{let } x = M \text{ in } R) \text{ in } E[y]$	$\mapsto$	$\text{let } x = M \text{ in } (\text{let } y = R \text{ in } E[y])$

Translation As for Let, except into Aff rather than Lin.

Fig. 6. The call-by-need lambda calculus Need.

(iii) *If* $M \underset{\mathsf{Need}}{\twoheadrightarrow} N$ *then* $M^* \underset{\mathsf{Aff}}{\twoheadrightarrow} N^*$.
(iv) $M \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\twoheadrightarrow} N$ *if and only if* $M^* \underset{\mathsf{Aff}}{\mapsto\!\!\!\!\twoheadrightarrow} N^*$.

Proof. The proof for Clauses (i)–(iii) is as for Let, noting that the reduction (G) in Need translates to (!Weakening) in Aff. For standard reduction we develop the reduction-closed grammars as usual, with the same erasure ‡ as for Let. □

Example 17 again applies as a counterexample to completeness for Clause (iii).

7. Equality and observational theories

Reduction describes how one term may be converted to another, but we often wish to relate pairs of terms where neither reduces to the other. In this section we consider two broader equivalence relations.

Equality is simply the least equivalence relation containing a given reduction relation. Let $\mathscr{T}$ be some reduction relation. Since all of the systems we consider are confluent, it is the case that $M =_{\mathscr{T}} N$ if and only if there is a L such that $M \underset{\mathscr{T}}{\twoheadrightarrow} L$ and $N \underset{\mathscr{T}}{\twoheadrightarrow} L$. Thus it is clear that any translation that is sound for reduction must also be sound for equality, and so the only question of interest is completeness.

Proposition 30 (Translation of equality). (i) $M =_{\mathsf{Name}} N$ *if and only if* $M^\circ =_{\mathsf{Lin}} N^\circ$.
(ii) *If* $M =_{\mathsf{Val}} N$ *then* $M^* =_{\mathsf{Lin}} N^*$.

(iii) *If* $M =_{\mathsf{Let}} N$ *then* $M^* =_{\mathsf{Lin}} N^*$.
(iv) *If* $M =_{\mathsf{Need}} N$ *then* $M^* =_{\mathsf{Aff}} N^*$.

Proof. In all cases, soundness of the translation for equality follows from soundness of the translation for reduction. For completeness of the standard translation we use the fact that for every Name term L and every Lin term M, if $L^\circ \underset{\mathsf{Lin}}{\twoheadrightarrow} M$, then there exists a Name term N such that $M \underset{\mathsf{Lin}}{\twoheadrightarrow} N^\circ$. In fact, by Lemmas 2(a) and 11(c) this N is just $M^\dagger$.

Given that the steadfast translations are not complete for reduction, it is not surprising that they are also incomplete for equality, but incompleteness of reduction does not fully account for the incompleteness of equality. For instance, when M does not reduce to a value in Val, the Val terms $(M\,N)$ and $((\lambda z.zN)M)$ are not necessarily equal in Val, but their translations are equal in Lin even though neither $(M\,N)^* \underset{\mathsf{Lin}}{\twoheadrightarrow} ((\lambda z.z\,N)M)^*$ nor $((\lambda z.z\,N)M)^* \underset{\mathsf{Lin}}{\twoheadrightarrow} (M\,N)^*$. The same example adopts to Let and Need. We return to the question of complete steadfast translations in the conclusion.

Certain pairs of terms which are not convertible to each other always exhibit the same behaviour in any context. The *observational equivalence* relations express this behaviour as a "black-box" attempt to distinguish terms.

Definition 31. Given a reduction theory $\mathscr{T}$ over a set of terms $\mathbf{S}$, two terms $M, n \in \mathbf{S}$ are observationally equivalent, written $M \approx_{\mathscr{T}} N$, if for all contexts C such that $C[M]$, $C[N] \in \mathbf{S}$ we have either that both $C[M]$ and $C[N]$ reduce to an answer, or that neither reduces to an answer.

When dealing with constants and base types, one usually imposes the additional requirement that if one context-wrapped term converges to a constant, then the other must converge to the same constant.

The intuitionistic reduction relations we have studied share observational equivalence theories:

Proposition 32. *Let* $M, N \in$ Name. *Then* $M \approx_{\mathsf{Name}} N$ *if and only if* $M \approx_{\mathsf{Need}} N$.

Proof. In previous work [24, 27]. □

Proposition 33. *Let* $M, N \in$ Val. *Then* $M \approx_{\mathsf{Val}} N$ *if and only if* $M \approx_{\mathsf{Let}} N$.

Proof. Immediate from conservative extension. □

Proposition 34. *The standard and steadfast translations of the various systems all preserve observational disequivalence.*
(i) *Let* $M, N \in$ Name. *If* $M^\circ \not\approx_{\mathsf{Lin}} N^\circ$ *then* $M \not\approx_{\mathsf{Name}} N$.
(ii) *Let* $M, N \in$ Val. *If* $M^* \not\approx_{\mathsf{Lin}} N^*$ *then* $M \not\approx_{\mathsf{Val}} N$.

(iii) *Let* $M,N \in \mathsf{Let}$. *If* $M^* \not\approx_{\mathsf{Lin}} N^*$ *then* $M \not\approx_{\mathsf{Let}} N$.
(iv) *Let* $M,N \in \mathsf{Need}$. *If* $M^* \not\approx_{\mathsf{Aff}} N^*$ *then* $M \not\approx_{\mathsf{Need}} N$.

Proof. For the first case we have

$$\begin{aligned} & M_0 \not\approx_{\mathsf{Name}} M_1 \\ & \quad \Rightarrow (\exists C)\ C[M_i] \underset{\mathsf{Name}}{\mapsto\!\!\!\!\rightarrow} A_0,\ (\forall A_1)\ C[M_{1-i}] \underset{\mathsf{Name}}{\mapsto\!\!\!\!\rightarrow} A_1 \\ & \quad \Rightarrow (\exists C)\ C[M_i]^\circ \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\rightarrow} A_0^\circ,\ (\forall A_1)\ C[M_{1-i}]^\circ \underset{\mathsf{Lin}}{\mapsto\!\!\!\!\rightarrow} A_1^\circ \\ & \quad \Rightarrow M_0^\circ \not\approx_{\mathsf{Lin}} M_1^\circ. \end{aligned} \tag{2}$$

We have the second condition of Eq. (2) by the completeness of the translation for standard reduction. The reasoning for the remaining cases is identical. □

We conjecture that the translations should also preserve observational equivalence. It is not clear how this result follows from the reduction theory; a proof based on the model theory is beyond the scope of this paper.

8. Extensions

In this section we discuss various extensions of our results. It is straightforward to extend the translations for constants and primitives (Section 8.1) and for products (Section 8.2), but an appropriate translation of sums is less clear (Section 8.3). Recursion does present some problems for call-by-need, but the translations of the call-by-name and call-by-value systems with recursion are well-behaved (Section 8.4). Finally, our results are easily transferred to an untyped framework (Section 8.5).

8.1. Constants and primitives

In addition to constants, all of the lambda calculi discussed may be straightforwardly extended by the inclusion of primitive applications $pM_1 \cdots M_k$ of arity k and a suitable reduction rule for each primitive.

$$(\delta) \quad pc_1 \cdots c_k \to \mathrm{apply}\,(p, c_1, \ldots, c_k)$$

Following Plotkin [30], 'apply' is a function that yields a closed term for a given primitive and constants.

We extend the translations as follows:

Call-by-name

$c^\circ \equiv c$
$(pM_1 \cdots M_k)^\circ \equiv pM_1^\circ \cdots M_k^\circ$

Call-by-value

$c^+ \equiv c$
$(pM_1 \cdots M_k)^* \equiv \mathrm{let}\ !x_1 = M_1^*\ \mathrm{in} \cdots \mathrm{let}\ !x_k = M_k^*\ \mathrm{in}\ px_1 \cdots x_k$

For the translation to be valid, the interpretation of primitives in the linear calculus must be related to the interpretations in both the call-by-name and call-by-value calculi:

$$\mathrm{apply}_{\mathsf{Lin}}(p, c_1, \ldots, c_k) \equiv (\mathrm{apply}_{\mathsf{Name}}(p, c_1, \ldots, c_k))^{\circ}$$

$$\mathrm{apply}_{\mathsf{Lin}}(p, c_1, \ldots, c_k) \equiv (\mathrm{apply}_{\mathsf{Val}}(p, c_1, \ldots, c_k))^{*}.$$

Again, the translation results carry through for the extended calculi.

8.2. Products

The extensions for products are straightforward:

Call-by-name

$(A \times B)^{\circ} \equiv A^{\circ} \& B^{\circ}$

$(M, N)^{\circ} \equiv (M^{\circ}, N^{\circ})$

$(\mathrm{fst}\ M)^{\circ} \equiv \mathrm{fst}\ M^{\circ}$

$(\mathrm{snd}\ M)^{\circ} \equiv \mathrm{snd}\ M^{\circ}$

Call-by-value

$(A \times B)^{+} \equiv A^{+} \& B^{+}$

$(V, W)^{+} \equiv (V^{+}, W^{+})$

$(\mathrm{fst}\ M)^{*} \equiv \mathrm{fst}\ (\mathrm{let}\ !z = M^{*}\ \mathrm{in}\ z)$

$(\mathrm{snd}\ M)^{*} \equiv \mathrm{snd}\ (\mathrm{let}\ !z = M^{*}\ \mathrm{in}\ z)$

In the call-by-name translation, & is the additive product of linear logic; we use the notations (M, N), $(\mathrm{fst}\ M)$ and $(\mathrm{snd}\ M)$ to stand both for $\times$ introduction and elimination in the intuitionistic lambda calculus and for & introduction and elimination in the linear lambda calculus. The call-by-value translation also uses this additive product. In contrast, Girard's version of the latter translation [14] defines

$$(A \times B)^{*} \equiv A^{*} \otimes B^{*},$$

which uses $\otimes$, the multiplicative (or tensor) product of linear logic. These two products are related by the isomorphism $!(A \& B) \simeq (!A) \otimes (!B)$. Thus our translation is isomorphic to Girard's, since

$$A^{*} \otimes B^{*} \equiv (!A^{+}) \otimes (!B^{+}) \simeq !(A^{+} \& B^{+}) \equiv !(A \times B)^{+} \equiv (A \times B)^{*}.$$

In call-by-value, the components of a pair are restricted to values. The more general construct (M, N) may be added to the call-by-value language by defining it as an abbreviation for $(\lambda x.\, \lambda y.\, (x, y))\, M\, N$. Restricting pairing to values makes the translation easier to define and corresponds to a restriction on pairs that arises naturally in call-by-need calculi [4, 20].

Syntactic domains As for Lin, plus the following:

Terms $L, M, N ::= \ldots | \mu x.M$

Typing judgements As for Lin, plus the following:

$$\frac{\Phi, x : A; - \vdash M : A}{\Phi; - \vdash \mu x.M : A}\mathsf{Rec}$$

Reduction relation As for Lin, plus the following:

(Rec) $\mu x.M \to M[x := \mu x.M]$

Fig. 7. The linear lambda calculus with recursion $\mathsf{Lin}_{\mathrm{rec}}$.

8.3. Sums

Extending the call-by-name translation to sums is straightforward. At the type level, following Girard, the translation is defined by

$$(A + B)^{\circ} \equiv (!A^{\circ}) \oplus (!B^{\circ}).$$

Here $\oplus$ is the additive (or direct) sum of linear logic; the term translation follows immediately.

Extending the call-by-value translation to sums is more problematic. At the type level, again following Girard, we would expect a definition satisfying the isomorphism

$$(A + B)^{*} \simeq A^{*} \oplus B^{*}.$$

As with products it is desirable to re-express the left-hand side in the form $!(A + B)^{+}$. Unfortunately, it is not clear how to choose a C such that $(!A) \oplus (!B) \simeq !C$; as a result the treatment of sums is less clear.

8.4. Recursion

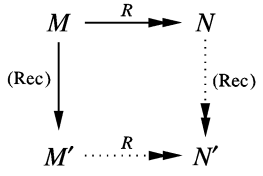
We can extend the linear lambda calculus for recursion on a single bound variable in the standard manner. We extend Lin as shown in Fig. 7 to obtain the recursive linear lambda calculus $\mathsf{Lin}_{\mathrm{rec}}$. The extension satisfies the same basic properties as the non-recursive system.

Proposition 35. $\mathsf{Lin}_{\mathrm{rec}}$ *satisfies subject reduction.*

Proposition 36. $\mathsf{Lin}_{\mathrm{rec}}$ *is confluent.*

Proof. Subject reduction (and the substitutivity result on which it depends) is straightforward. The extension of confluence from the non-recursive to the recursive calculus is also clear. In general, for showing confluence of any of the extensions of our systems, we rely on two facts: First, it follows trivially from the confluence of R on R-terms that

R is also confluent on R_{rec}-terms. Second, for any system R and its recursive extension R_{rec}, taking terms ranging over R_{rec}, we can show



by extending the non-recursive system's marking scheme to track unwindings of the recursion operator as well. This relationship is stronger than weak confluence, and full confluence does follow. From these two facts, confluence for any of the recursive systems follows.

Figs. 8 and 9 present simple extensions to the call-by-name and call-by-value calculi for recursion. In the call-by-value extension, it is somewhat surprising to see the term $\mu x.\lambda y.M$ as a value, since it is also a redex. This detail is necessary for the properties of the translation to behave, and is not especially egregious as all such values are always only one (Rec) step away from being a non-redex value. These systems again satisfy the same basic properties of the non-recursive ones:

Proposition 37. *Both* $\mathsf{Name}_{\mathrm{rec}}$ *and* $\mathsf{Val}_{\mathrm{rec}}$ *have the subject reduction property.*

Proposition 38. *Both* $\mathsf{Name}_{\mathrm{rec}}$ *and* $\mathsf{Val}_{\mathrm{rec}}$ *are confluent.*

Extending the translations for $\mathsf{Name}_{\mathrm{rec}}$ and $\mathsf{Val}_{\mathrm{rec}}$ is straightforward. Both extended translations are sound and complete for the respective calculi with recursion.

Proposition 39. *The translation* $\circ$ *from* $\mathsf{Name}_{\mathrm{rec}}$ *to* $\mathsf{Lin}_{\mathrm{rec}}$ *preserves substitutions, preserves typing judgements and preserves reduction sequences*: *For* $M,N \in \mathsf{Name}_{\mathrm{rec}}$,

(i) $(M[x:=N])^\circ \equiv M^\circ[x:=N^\circ]$.

(ii) $\Gamma \vdash_{\mathsf{Name}_{\mathrm{rec}}} M:A$ *if and only if* $\Gamma^\circ; - \vdash_{\mathsf{Lin}_{\mathrm{rec}}} M^\circ : A^\circ$.

(iii) $M \underset{\mathsf{Name}_{\mathrm{rec}}}{\twoheadrightarrow} N$ *if and only if* $M^\circ \underset{\mathsf{Name}_{\mathrm{rec}}}{\twoheadrightarrow} N^\circ$.

Proof. Clauses (i) and (ii) are as before. For soundness in Clause (iii) we take

$$\mu x.M \underset{\mathsf{Name}_{\mathrm{rec}}}{\longrightarrow} M[x:=\mu x.M],$$

and in $\mathsf{Lin}_{\mathrm{rec}}$ we have

$$\begin{aligned}&(\mu x.M)^\circ\\ &\equiv \mu x.M^\circ\\ &\underset{(\mathrm{Rec})}{\longrightarrow} M^\circ[x:=\mu x.M^\circ]\\ &\equiv (M[x:=\mu x.M)^\circ.\end{aligned}$$

Syntactic domains As for Name, plus the following:

Terms $L, M, N ::= \ldots | \mu x.M$

Typing judgements As for Name, plus the following:

$$\frac{\Gamma, x : A; - \vdash M : A}{\Gamma \vdash \mu x.M : A}\mathsf{Rec}$$

Reduction relation As for Name, plus the following:

(Rec) $\mu x.M \to M[x := \mu x.M]$

Standard translation As for Name into Lin, plus the following:

$(\mu x.M)^\circ \equiv \mu x.(M^\circ)$

Fig. 8. The call-by-name lambda calculus with recursion $\mathsf{Name}_{\mathrm{rec}}$.

Syntactic domains As for Val, plus the following:

Values $V, W ::= \ldots | \mu x.\lambda y.M$

Typing judgements As for Val, plus the following:

$$\frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \mu f.\lambda x.M : A \to B}\mathsf{Rec}$$

Reduction relation As for Val, plus the following:

(Rec) $\mu x.\lambda y.M \to (\lambda y.M)[x := \mu x.\lambda y.M]$

Steadfast translation As for Val into Lin, plus the following:

$(\mu f.\lambda x.M)^+ \equiv \mu f.\lambda z.$ let $!x = z$ in M^*

Fig. 9. The call-by-value lambda calculus with recursion $\mathsf{Val}_{\mathrm{rec}}$.

Completeness is also easy; we extend the grammar of recursion-closed $\circ$-images by

$S ::= \cdots | \mu x.S,$

and the map $\dagger$ by

$(\mu x.S)^\dagger \equiv \mu x.S^\dagger.$

Clearly $\dagger$ is again right-inverse to $\circ$, and preserves the (Rec) step.

Proposition 40. *The translation* $*$ *from* $\mathsf{Val}_{\mathrm{rec}}$ *to* $\mathsf{Lin}_{\mathrm{rec}}$ *preserves substitution of values, preserves typing judgements and preserves reduction sequences*: *For* $M,N,V \in \mathsf{Val}_{\mathrm{rec}}$,

(i) $(M[x := V])^\circ \equiv M^*[x := V^+]$.

(ii) $\Gamma \vdash_{\mathsf{Val}_{\mathrm{rec}}} M : A$ *if and only if* $\Gamma^*; - \vdash_{\mathsf{Val}_{\mathrm{rec}}} M^* : A^*$.

(iii) *If* $M \underset{\mathsf{Val}_{\mathrm{rec}}}{\twoheadrightarrow} N$ *then* $M^* \underset{\mathsf{Lin}_{\mathrm{rec}}}{\twoheadrightarrow} N^*$.

Proof. Clauses (i) and (ii) are as usual. For soundness in Clause (iii) we take $\mu f.\lambda x.M \underset{\mathsf{Val}_{\mathrm{rec}}}{\twoheadrightarrow} \lambda x.M[f := \mu f.\lambda x.M]$, and then in $\mathsf{Lin}_{\mathrm{rec}}$ we have

$$
\begin{aligned}
&!\mu f.\lambda z.\ \text{let } !xz \text{ in } M^* \\
&\quad \underset{(\mathrm{Rec})}{\longrightarrow} !(\lambda z.\ \text{let } !x = z \text{ in } M^*)[f := \mu f.\lambda z.\ \text{let } !x = z \text{ in } M^*] \\
&\quad \equiv (\lambda x.M[f := \mu f.\lambda x.M])^*. \qquad \square
\end{aligned}
$$

In our earlier work [26] we had conjectured that two different recursion operators were possible for $\mathsf{Lin}_{\mathrm{rec}}$: the one used above, and in addition a second of type

$$\frac{\Phi, x : A; - \vdash M : !A}{\Phi; - \vdash \mu_v x.M : !A}\mathrm{Rec}_{\mathsf{v}},$$

or equivalently

$$\frac{\Phi; x : !A \vdash M : !A}{\Phi; - \vdash \mu_v x.M : !A}\mathrm{Rec}_{\mathsf{v}}.$$

The motivation for two systems is that one rule – the one we use here in $\mathsf{Lin}_{\mathrm{rec}}$ – seems to correspond to the standard translation of the call-by-name Rec rule, while the Rec_v rule above seems to correspond to the steadfast translation of the call-by-value Rec rule. In fact, the two seem to be equivalent.

The Rec_v typing rule gives rise to the $\mathsf{Lin}_{\mathrm{rec}v}$ calculus of Fig. 10. We can map terms from one recursive linear calculus into the other with the maps $\flat$ and $\sharp$ of Fig. 11. We find that the two maps are inverses of each other modulo convertibility.

Syntactic domains As for $\mathsf{Lin}_{\mathrm{rec}}$.

Typing judgements As for Lin, plus the following:

$$\frac{\Phi, x : A; - \vdash M : !A}{\Phi; - \vdash \mu x.M : !A}\mathrm{Rec}_{\mathsf{v}}$$

Reduction relation As for Lin, plus the following:

$$(\mathrm{Rec}_v) \quad \mu x.M \to M[x := \text{let } !y\mu x.M \text{ in } y]$$

Fig. 10. The alternate linear lambda calculus with recursion $\mathsf{Lin}_{\mathrm{rec}\,v}$.

From $\mathsf{Lin}_{\mathrm{rec}}$ **to** $\mathsf{Lin}_{\mathrm{rec}\,v}$

$$
\begin{aligned}
x^{\flat} &\equiv x \\
(\lambda x.M)^{\flat} &\equiv \lambda x.M^{\flat} \\
(M\,N)^{\flat} &\equiv M^{\flat}\,N^{\flat} \\
(\text{let}\ \ x=M\ \text{in}\ N)^{\flat} &\equiv \text{let}\ x=M^{\flat}\ \text{in}\ N^{\flat} \\
(\mu x.M)^{\flat} &\equiv \text{let}\ !y=\mu_v x.!M^{\flat}\ \text{in}\ y
\end{aligned}
$$

From $\mathsf{Lin}_{\mathrm{rec}v}$ **to** $\mathsf{Lin}_{\mathrm{rec}}$

$$
\begin{aligned}
x^{\sharp} &\equiv x \\
(\lambda x.M)^{\sharp} &\equiv \lambda x.M^{\sharp} \\
(M\,N)^{\sharp} &\equiv M^{\sharp}\,N^{\sharp} \\
(\text{let}\ \ x=M\ \text{in}\ N)^{\sharp} &\equiv \text{let}\ \ x=M^{\sharp}\ \text{in}\ N^{\sharp} \\
(\mu_v x.M)^{\sharp} &\equiv !\mu x.\ \text{let}\ !z=M^{\sharp}\ \text{in}\ z
\end{aligned}
$$

Fig. 11. Maps between the alternate recursive linear lambda calculi.

Starting from $\mathsf{Lin}_{\mathrm{rec}}$ the result is simple:

Proposition 41. *Let* $M \in \mathsf{Lin}_{\mathrm{rec}}$. *Then* $M^{\flat\sharp} \underset{\mathsf{Lin}_{\mathrm{rec}}}{\twoheadrightarrow} M$.

Proof. All cases are trivial aside from the recursion binding:

$$
\begin{aligned}
&(\mu x.M)^{\flat\sharp} \\
&\quad\equiv \text{let}\ !y=!\mu x.\ \text{let}\ !z=!M^{\flat\sharp}\ \text{in}\ z\ \text{in}\ y \\
&\quad\underset{\beta!}{\longrightarrow} \mu x.\ \text{let}\ !z=!M^{\flat\sharp}\ \text{in}\ z \\
&\quad\underset{\beta!}{\longrightarrow} \mu x.M^{\flat\sharp}. \qquad \square
\end{aligned}
$$

Unfortunately, when we begin with a term in $\mathsf{Lin}_{\mathrm{rec}\,v}$ the development is less clear. The image under the composition of the two maps is then convertible only if we also consider an eta reduction rule ($¡\eta$),

($¡\eta$) $\quad !\text{let}\ !x=M\ \text{in}\ x \to M.$

Proposition 42. *Let* $M \in \mathsf{Lin}_{\mathrm{rec}\,v}$. *Then* $M^{\flat\sharp} \underset{(¡\eta)}{\twoheadrightarrow} M$.

Proof. All cases are again trivial aside from the recursion binding:

$$
\begin{aligned}
&(\mu_v x.M)^{\sharp\flat}\\
&\quad \equiv !\text{let } !y = (\mu_v x.\ !\text{let } !z = M^{\sharp\flat} \text{ in } z) \text{ in } y\\
&\quad \xrightarrow[(¡\eta)]{} \mu_v x.\ !\text{let } !z = M^{\sharp\flat} \text{ in } z\\
&\quad \xrightarrow[(¡\eta)]{} \mu_v x.M^{\sharp\flat}. \qquad \square
\end{aligned}
$$

It is also easy to verify that $\flat$ is sound for reduction without the $(¡\eta)$ rule, and that $\sharp$ is sound for equality with the $(¡\eta)$ rule is $\mathsf{Lin}_{\mathrm{rec}}$. So while the equivalence between the systems as they are may not be perfect, it seems to be the case that by enriching the linear calculi with η and perhaps other rules, it might be strengthened. Unfortunately, adding η to the intuitionistic calculi poses other problems, which we discuss below in Section 9.1; in particular the rule $(¡\eta)$ is rather unusual among η rules for linear systems. In any event, although the soundness and completeness results of Propositions 39 and 40 do not transfer directly to analogous results for translations into $\mathsf{Lin}_{\mathrm{rec}v}$, these alternate translations are sound and complete in the same manner as the corresponding non-recursive systems, which can be demonstrated in the usual way, extending the translations by

$$(\mu x.M)^{\bar{\circ}} \equiv \text{let } !y = \mu x.!M^{\bar{\circ}} \text{ in } y$$

$$(\mu f.\lambda x.M)^{\bar{\dagger}} \equiv \text{let } !z = (\mu f.!\lambda y.\ \text{let } !x = y \text{ in } M^{\bar{*}}) \text{ in } z$$

and taking

$$S ::= \cdots \mid \text{let } !y = \mu x.!S \text{ in } y$$

in the proof of completeness of the extended map from $\mathsf{Name}_{\mathrm{rec}}$ to $\mathsf{Lin}_{\mathrm{rec}\,v}$.

Braüner [13] has studied the properties of an extension for recursion of the standard translation from Name to Lin in models for these systems. His formulation is essentialy the same as our $\mathsf{Lin}_{\mathrm{rec}}$ calculus, although since we (1) separate linear and intuitionistic assumptions and (2) avoid term structures corresponding to structural rules, while he does neither, the syntaxes of our respective systems appear quite different at first glance. Braüner comments that the standard translation from call-by-name into his system is sound for reduction, but addresses neither completeness nor call-by-value systems and the steadfast translation. Rather, Braüner is concerned mainly with the categorical models of recursion in the two systems $\mathsf{Name}_{\mathrm{rec}}$ and $\mathsf{Lin}_{\mathrm{rec}}$.

This form of recursion is only loosely based on what one would find in actual programming languages. Rather than recursion over a single variable, one uses multiple, mutually recursive bindings (letrec $x_1 = M_1, \ldots, x_k = M_k$ in N). This structure is especially relevant in call-by-need, where single recursion would cause excessive duplication. On the surface, this change appears to pose a problem only because the expected increase in the size of the calculus is awkward, but in fact with letrec one

Syntactic domains As for Lin, but without types.
Well-formedness judgements

$$\frac{}{-;x\vdash x}\mathsf{Id} \qquad \frac{\Phi,y,z;\Gamma\vdash M}{\Phi,x;\Gamma\vdash M[y:=x,z:=x]}\mathsf{Contraction}$$

$$\frac{\Phi;\Gamma\vdash M}{\Phi,x;\Gamma\vdash M}\mathsf{Weakening} \qquad \frac{\Phi;x,\Gamma\vdash M}{\Phi,x;\Gamma\vdash M}\mathsf{Dereliction}$$

$$\frac{\Phi;-\vdash M}{\Phi;-\vdash !M}!\text{-I} \qquad \frac{\Phi;\Gamma\vdash M \quad \Psi,x;\Delta\vdash N}{\Phi,\Psi;\Gamma,\Delta\vdash \text{let } !x=M \text{ in } N}!\text{-E}$$

$$\frac{\Phi;\Gamma,x\vdash M}{\Phi;\Gamma\vdash \lambda x.M}\multimap\text{-I} \qquad \frac{\Phi;\Gamma\vdash M \quad \Psi;\Delta\vdash N}{\Phi,\Psi;\Gamma,\Delta\vdash MN}\multimap\text{-E}$$

Reduction relation As for Lin.

Fig. 12. The untyped linear lambda calculus $\mathsf{Lin}_{\mathrm{untyp}}$.

must restrict the selection of redexes, or otherwise the calculus will not be confluent. Ariola and Klop [5] first pointed out this problem; Ariola and Felleisen [3] described a confluent call-by-need system, and Ariola and Blom [2] gave a general description of cyclic lambda terms. Launchbury [20] explained call-by-need with letrec's using *natural semantics* rather than reduction; his system prescribes a fixed reduction order which includes the restrictions described by Ariola and Felleisen. It is not immediately clear how to adapt our results for a recursive Need system, nor for Let beyond the single-recursive bindings of $\mathsf{Val}_{\mathrm{rec}}$.

8.5. Untyped languages

The confluence and translation results do not depend on types, and so carry through directly for untyped version of these calculi. The only trickiness is that we have used the type rules of the linear lambda calculus to enforce the constraint that variables bound by linear assumptions appear linearly, and that all free variables of a term $!M$ be bound by intuitionistic assumptions. It is easy to enforce the same constraints without recourse to types, for instance by using well-formedness judgements. We present such a system, $\mathsf{Lin}_{\mathrm{untyp}}$, in Fig. 12. Our earlier results for Lin do not depend on types, and we have analogues of the substitution and subject reduction properties, plus confluence as well:

Lemma 43. *Well-formedness is preserved by substitution of well-formed terms*:
(i) *Let* $\Phi;\Gamma,x\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M$ *and* $\Psi;\Delta\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} N$. *Then* $\Phi,\Psi;\Gamma,\Delta\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M[x:=N]$.
(ii) *Let* $\Phi,x;\Gamma\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M$ *and* $\Psi;-\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} N$. *Then* $\Phi,\Psi;\Gamma\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M[x:=N]$.

Proposition 44. *If* $\Phi;\Gamma\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M$ *and* $M \underset{\mathsf{Lin}_{\mathrm{untyp}}}{\longrightarrow} N$ *then* $\Phi;\Gamma\vdash_{\mathsf{Lin}_{\mathrm{untyp}}} N$.

Proposition 45. $\mathsf{Lin}_{\mathrm{untyp}}$ *is confluent.*

Similar results also hold for an untyped Aff, and similar substitution and confluence results also hold for the other systems: for $\mathsf{Name}_{\mathrm{untyp}}$ and $\mathsf{Val}_{\mathrm{untyp}}$ the results are standard; for $\mathsf{Let}_{\mathrm{untyp}}$ and $\mathsf{Need}_{\mathrm{untyp}}$ the results can be found in our previous work on call-by-need [4, 27].

We can use the translations of the typed systems as they are for their untyped counterparts. For example, for $\mathsf{Name}_{\mathrm{untyp}}$ we have

Proposition 46 (Untyped call-by-name translation). *The translation* $\circ$ *from* $\mathsf{Name}_{\mathrm{untyp}}$ *to* $\mathsf{Lin}_{\mathrm{untyp}}$ *produces well-formed terms and preserves reductions*:
(i) Γ *contains the free variables of* M *if and only if* $\Gamma^{\circ}; - \vdash_{\mathsf{Lin}_{\mathrm{untyp}}} M^{\circ}$.
(ii) $M \underset{\mathsf{Name}_{\mathrm{untyp}}}{\twoheadrightarrow} N$ *if and only if* $M^{\circ} \underset{\mathsf{Name}_{\mathrm{untyp}}}{\twoheadrightarrow} N^{\circ}$.

Proof. Clause (ii) is exactly as in the typed case. Clause (i) follows since we can write trivial "free variables on the left of $\vdash$" judgements for $\mathsf{Name}_{\mathrm{untyp}}$ which are again just Name typing rules with the types removed. The corresponding "translation" of these free variable lists corresponds to the translation of typing environments in the typed system, and the result follows as before.

9. Conclusions

We have shown that call-by-name, call-by-value, call-by-let and call-by-need can be explained by translations into linear and affine lambda calculi. These transformations begin to provide a logical explanation of call-by-value and call-by-need in the style of the Curry-Howard isomorphism. Many interesting questions remain, and we conclude this paper by mentioning two below.

9.1. Eta rules

It is common to include an $(\eta\rightarrow)$ rule in the call-by-name calculus, and an $(\eta\rightarrow_v)$ rule in the call-by-value calculus.

$$(\eta\rightarrow) \quad \lambda x.(Mx) \rightarrow M \quad \text{if } x \text{ not free in } M,$$

$$(\eta\rightarrow_v) \quad \lambda x.(Vx) \rightarrow V \quad \text{if } x \text{ not free in } V.$$

However it is not clear how one might formulate η rules in Lin in a way which admits useful results about the translations. Sabry and Wadler [34] identify a variant of Lin including η rules into which Moggi's computation lambda calculus [28, 29] (see the next section below) can be translated by a map which is in fact a reflection. However, Sabry and Wadler's target linear calculus does not allow $(!\multimap)$ steps, restricts the form of β redexes, and uses an awkward $(\eta\multimap)$ rule

$$(\eta\multimap) \quad \lambda x.\ .\text{let } !y = x \text{ in } V!y \rightarrow V, \quad y \notin \mathrm{fv}(V)$$

Syntactic domains

Types	A, B, C	$::= Z \mid A \to B$
Terms	L, M, N	$::= V \mid E$
Values	V	$::= x \mid \lambda x.M$
Non-values	E	$::= MN \mid \text{let } x = M \text{ in } N$

Typing judgements As for Let.

Reduction relation

(β_v)	$(\lambda x \cdot M)V$	$\to M[x := V]$
(η_v)	$\lambda x \cdot (Vx)$	$\to V$, if x not free in V
(let_v)	let $x = V$ in M	$\to M[x := V]$
(id)	let $x = M$ in x	$\to M$
(comp)	let $y = (\text{let } x = L \text{ in } M)$ in N	$\to$ let $x = L$ in (let $y = M$ in N)
(let.1)	EM	$\to$ let $z = E$ in zM
(let.2)	VE	$\to$ let $x = E$ in Vx

Fig. 13. Moggi's computational lambda calculus Comp.

for a restriction V of linear terms to variables or certain abstractions. Various other rules seem possible, but it is not clear how one might construct a simple system whose rules have the same logical resonance as the β reduction axioms considered in this paper.

9.2. Complete steadfast translations and Moggi's computational lambda calculus

It is reasonable to ask whether more laws could be added to Val, Let or Need so that the corresponding steadfast translations would be complete for either reduction or equality. A hint as to a suitable extension is provided by Moggi's computational lambda calculus Comp [28, 29], shown in Fig. 13. (The version of the calculus shown here is based on the untyped reduction calculus, which Moggi calls λ_c, and which appears in his technical report [28] but not his LICS paper [29].) The terms of this theory are the same as for Let; though the grammar now distinguishes non-values E as well as values V. This system satisfies subject reduction, and Moggi shows that it is confluent. The system is designed so that it is strongly normalising even without types, apart from rule (β_v).

It is not hard to show that the equalities of Let are properly contained in the equalities of Comp; that is, $M =_{\mathsf{Let}} N$ implies $M =_{\mathsf{Comp}} N$, but not conversely. Furthermore, the reductions of Let and the reductions of Comp are incomparable; that is, $M \twoheadrightarrow_{\mathsf{Let}} N$ does not imply $M \twoheadrightarrow_{\mathsf{Comp}} N$, nor is the converse true. It is interesting open question whether there is an extension of Let that has the same equalities as Comp. It is a further

interesting question to know if there is an extension of Lin such that the encoding of the extended Let into the extended Lin via $*$ is sound and complete.

This question brings us full circle. Plotkin showed that the continuation-passing style translation from Val into itself is sound but not complete [30]. Moggi designed Comp to be sound and complete for the monad translation (which generalises CPS) [28], and Sabry and Felleisen verified that the CPS translation from Comp into Val is both sound and complete [33], thus answering the question implicitly raised by Plotkin. The questions raised here about complete extensions of Val and Let are in a similar vein. It is not clear if the translations into linear lambda calculus described here will have the same value as the translations into continuation-passing style described by Plotkin. But if our answers are not as good, perhaps we can at least claim to be asking the right sort of questions!

References

[1] S. Abramsky, Computational interpretations of linear logic, Theoret. Comput. Sci. 111 (1993) 3–57.

[2] Z.M. Ariola, S. Blom, Cyclic Lambda calculi, Proc. TACS'97, Conf. on Theoretical Aspects of Computer Science, Sendai, Japan, 1997.

[3] Z.M. Ariola, M. Felleisen, The call-by-need lambda calculus, J. Funct. Programm. 7 (1997); see also Tech. Report CIS-TR-94-23, Department of Computer Science, University of Oregon, October 1994.

[4] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, P. Wadler, A call-by-need lambda calculus, Conf. Rec. POPL'95: 22nd ACM Symp. on Principles of Programming Languages, ACM Press, San Francisco, CA, January 1995, pp. 233–246.

[5] Z.M. Ariola, J.W. Klop, Cyclic lambda graph rewriting, Proc. LICS'94: IEEE Conf. on Logic in Computer Science, Paris, France, 1994.

[6] Andrew Barber, DILL – dual intuitionistic linear logic, Draft Paper, Dual intuitionistic linear logic, October 1995, to appear.

[7] H.P. Barendregt, The Lambda Calculus: its Syntax and Semantics, Studies in Logic and the Foundations of Computer Science, vol. 103, North-Holland, Amsterdam, 1981.

[8] H.P. Barendregt, K. Hemerik, Types in Lambda calculus and programming languages, Proc. ESOP'90: European Symp. on Programming, Lecture Notes in Computer Science, vol. 432, Springer, Berlin, 1990, pp. 1–35.

[9] P.N. Benton, Strong normalisation for the linear term calculus, J. Funct. Programm. 5 (1) (1995) 65–80.

[10] N. Benton, G. Bierman, V. de Paiva, M. Hyland, Type assignment for intuitionistic linear logic, Tech. Report 262, Computing Laboratory, University of Cambridge, August 1992.

[11] G. Bierman, On Intuitionistic Linear Logic, Ph.D. Thesis, University of Cambridge Computer Laboratory, December 1993; also appears as Tech. Report 346, Computing Laboratory, University of Cambridge, August 1994.

[12] G. Bierman, What is a categorical model of intuitionistic linear logic? Proc. 2nd Internat. Conf. on Typed Lambda Calculus, Lecture Notes in Computer Science, vol. 902, Edinburgh, Scotland, Springer, Berlin, April 1995.

[13] T. Braüner, The Girard translation extended with recursion, Proc. CSL'94: 8th Workshop on Computer Science Logic, Lecture Notes in Computer Science, vol. 933, Kazimierz, Poland, Springer, Berlin, September 1994; extended report in: Tech. Report RS-95-13, BRICS Group, Dept. of Computer Science, University of Aarhus, February 1995.

[14] J.-Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1987) 1–102.

[15] J.-Y. Girard, On the unity of logic, Ann. Pure Appl. Logic 59 (1993) 201–217.

[16] G. Gonthier, M. Abadi, J.-J. Levy, The geometry of optimal lambda reduction, Conf. Rec. POPL'92: 18th ACM Sym on Principles of Programming Languages, ACM Press, Albuquerque, New Mexico, January 1992.

[17] S. Holmström, A linear functional language, Draft Paper, Chalmers University of Technology, 1988.

[18] B. Jacobs, Semantics of weakening and contraction, Ann. Pure Appl. Logic 69 (1994) 73–106.
[19] Y. Lafont, The linear abstract machine, Theoret. Comput. Sci. 59 (1988) 157–180.
[20] J. Launchbury, A natural semantics for lazy evaluation, Conf. Rec. POPL'93: 19th ACM Symp. on Principles of Programming Languages, ACM Press, Charleston, January 1993.
[21] P. Lincoln, J. Mitchell, Operational aspects of linear lambda calculus, Proc. LICS'92: 7th IEEE Symp. on Logic in Computer Science, IEEE Press, Santa Cruz, California, June 1992.
[22] I. Mackie, Lilac: a functional programming language based on linear logic, Master's Thesis, Imperial College London, 1991.
[23] I. Mackie, The geometry of implementation, Ph.D. Thesis, Imperial College London, 1994.
[24] J. Maraist, Comparing reduction strategies in resource-conscious lambda calculi, Doctoral Thesis, University of Karlsruhe, Germany, 1997.
[25] J. Maraist, Separating weakening and contraction a linear lambda calculus, Proc CATS'98, Computing: the 4th Australian Theory Symp., Perth, February 1998.
[26] J. Maraist, M. Odersky, D.N. Turner, P. Wadler, Call-by-name, call-by-value, call-by-need and the linear lambda calculus, Proc. MFPS'95: 11th Conf. on the Mathematical Foundations of Programming Semantics, Elsevier, ENTCS 1, New Orleans, March/April 1995.
[27] J. Maraist, M. Odersky, P. Wadler, The call-by-need lambda calculus, J. Funct. Programm. 8(3) 1998 275–317.
[28] E. Moggi, Computational lambda-calculus and monads, Tech. Report ECS-LFCS-88-66, Laboratory for the Foundations of Computer Science, University of Edinburgh, October 1988.
[29] E. Moggi, Computational lambda-calculus and monads, Proc. LICS'89: 4th IEEE Symp. on Logic in Computer Science, IEEE Press, Asilomar, California, June 1989.
[30] G.D. Plotkin, Call-by-name, call-by-value and the λ calculus, Theoret. Comput. Sci. 1 (1975) 125–159.
[31] J.C. Reynolds, The discoveries of continuations, Lisp Symbolic Comput. 6(3/4) (1993) 233–248.
[32] S.R. della Rocca, L. Roversi, Lambda calculus and intuitionistic linear logic, manuscript, July 1994. Available from L. Roversi, University of Pisa, rover@di.unipi.it.
[33] A. Sabry, M. Felleisen, Reasoning about programs in continuation-passing style, Lisp Symbolic Comput. 6(3/4) (1993) 289–360.
[34] A. Sabry, P. Wadler, A reflection on call-by-value, ACM Trans. Programm. Languages 19(6) (1997) 916–941. (earlier version appears in: ACM International Conference on Functional Programming, ACM Press, Philadelphia, May 1996).
[35] H. Schellinx, The noble art of linear decorating, Ph.D. Dissertation, Institute for Logic, Language, and Computation, University of Amsterdam, 1994.
[36] R.A.G. Seely, Linear logic, $*$-autonomous categories, and cofree coalgebras, Categories in Computer Science and Logic, AMS Contemporary Mathematics, vol. 92, June 1989.
[37] A.S. Troelstra, Lectures on Linear Logic, CSLI Lecture Notes, 1992.
[38] D.N. Turner, P. Wadler, C. Mossin, Once upon a type, Conf. Rec. FPCA'95: 7th ACM Symp. on Functional Programming and Computer Architecture, ACM Press, San Diego, CA, June 1995.
[39] P. Wadler, Linear types can change the world!, in: M. Broy, C. Jones (Eds.), Programming Concepts and Methods, North-Holland, Sea of Galilee, Israel, April 1990.
[40] P. Wadler, Is there a use for linear logic? Proc. PEPM'91: ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, New Haven, Connecticut, June 1991.
[41] P. Wadler, A syntax for linear logic, Proc. MFPS'93: 9th Internat. Conf. on the Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science, vol. 802, New Orleans, Louisiana, Springer, Berlin, April 1993.
[42] P. Wadler, A taste of linear logic, Proc. MFCS'93: Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. 711, Gdansk, Poland, Springer, Berlin, August 1993.